INCAST 2008- 128

ASSESSMENT OF SIMD PROGRAMMING ON GRAPHICS CARD

S.Nagendran[1], Mohammad Ashrafulla.K[2], Jagadevi Bakka[3]
[1] Flosolver unit, National Aerospace Laboratories, Bangalore, India, nag@flosolver.nal.res.in
[2] Dept of Computer Science & Engg, AMC Engineering College, Bangalore, India, ashraf_zyx@yahoo.com
[3] Dept of Information Science & Engg, MSRIT, Bangalore, India, jagadevi.bakka@gmail.com

ABSTRACT: Recent advance of the technologies incorporated in graphics hardware has enabled general-purpose computations on graphics hardware, which can further be used for high-performance computation in low cost. Over the past six years GPUs have shown a marked increase in its performance and capabilities. Modern GPUs are now fully programmable massively parallel floating point processors demonstrating a performance/cost ratio superior to central processing units (CPUs) with computations of high arithmetic intensity. This effort in general-purpose computing on the GPU, also known as GPU computing, has positioned the GPU as a compelling alternative to traditional microprocessors in high-performance computer systems of the future. GPU is massively multithreaded parallel computing SIMD platform and NVIDIA is best known for a line of outstanding graphics processors that have become popular as the basis for graphics cards. Modern NVIDIA GPUs are not single processors but rather are parallel supercomputers on a chip that consist of very many, very fast processors. NVIDIA has supported this trend by releasing the CUDA (Compute Unified Device Architecture) interface library to allow intrepid applications developers to write code that can be uploaded for execution by NVIDIA's massively parallel GPUs. In this paper we analyze the performance of SIMD applications which are accelerated by GPU comparing that to their CPU counterparts. We initially design a set of SIMD programs using C language which are executed on the CPU, the same programs are data parallelized with the help of CUDA API and executed with GPU to find the performance gain and thereby analyze what potential benefits GPU can offer.

1. INTRODUCTION

1.1 An Overview of GPU

GPU stands for Graphics Processing Unit and is a single chip processor used primarily for 3D applications. It creates lighting effects and transforms objects every time a 3D scene is redrawn. These are mathematically-intensive tasks, which otherwise, would put quite a strain on the CPU. Lifting this burden from the CPU frees up cycles that can be used for other jobs.

Today the programmable graphics processor unit has evolved into an absolute computing workhorse with multiple cores driven by very high memory bandwidth. The main reason behind such an evolution is that the GPU is specialized for compute-intensive, highly parallel computation, exactly what graphics rendering is about and therefore is designed such that more transistors are devoted for data processing rather than control flow.

Till now, however, accessing all that computational power packed into the GPU and efficiently leveraging it for non-graphics applications remained tricky, mainly due to following reasons:

- The GPU could only be programmed through a graphics API, imposing a high learning curve to the novice and the overhead of an inadequate API to the non-graphics application.
- The GPU DRAM could be read in a general way (GPU programs can gather data elements from any part of DRAM), but could not be written in a general way (GPU programs cannot scatter information to any part of DRAM), removing a lot of the programming flexibility readily available on the CPU.
- Some applications were bottlenecked by the DRAM memory bandwidth, under-utilizing the GPU's computational power.

CUDA is answers to these problems and exposes GPU as a truly generic data-parallel computing device.

1.2 CUDA

CUDA stands for Compute Unified Device Architecture and is a new software and hardware architecture for issuing and managing computations on the GPU as a data parallel computing device without the need of mapping them to a graphics API. CUDA has been developed by Nvidia and to use this architecture requires an Nvidia GPU. It is available for the GeForce 8 series GPUs, Tesla Solutions and some Quadro Solutions.

CUDA solves the earlier problems of GPU:

- CUDA provides an API that's an extension to the C programming language for a minimum learning curve.
- CUDA provides general DRAM memory addressing for more programming flexibility i.e. both scatter and gather memory operations.
- CUDA features a parallel data cache or on-chip shared memory with very fast general read and write access, that threads use to share data with each other.

2. CUDA HARDWARE MODEL

The device (GeForce 8800 GTX) is implemented as a set of multiprocessors, 16 multiprocessors with 8 processors each (figure 1). Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD): At any given clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data. Each multiprocessor has its own shared memory which is common to all the 8 processors inside it. It also has a set of 32-bit registers, texture, and constant memory caches. Communication between multiprocessors is through the device memory, which is available to all the processors of the multiprocessors.

3. CUDA PROGRAMMING MODEL

When programmed through CUDA, the GPU is viewed as a compute device capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU. A portion of an application that is executed many times, but independently on different data, can be isolated into a function that is executed on the device as many different threads. To that effect, such a function is compiled to the instruction set of the device and the resulting program, called a kernel, is downloaded to the device. Both the host and the device maintain their own DRAM, referred to as host memory and device memory, respectively. The batch of threads that executes a kernel is organized as a grid of thread blocks (figure 2). A thread block is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses. There is a limit on the maximum number of threads that a block can contain. However, blocks of same dimensionality and size that execute the same kernel can be batched together into a grid of blocks, so that the total number of threads that can be launched in a single kernel invocation is much larger. This comes at the expense of reduced thread cooperation, because threads in different thread blocks from the same grid cannot communicate and synchronize with each other.

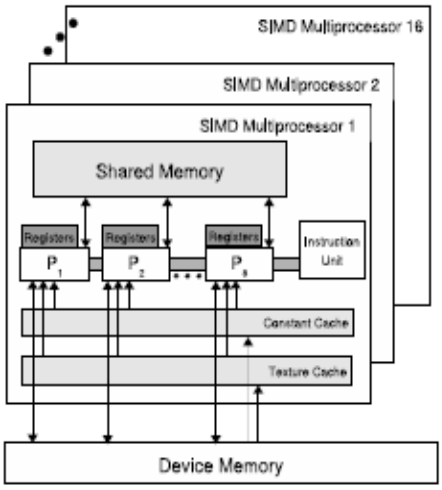

Figure 1: Hardware Model
(Source: Nvidia)

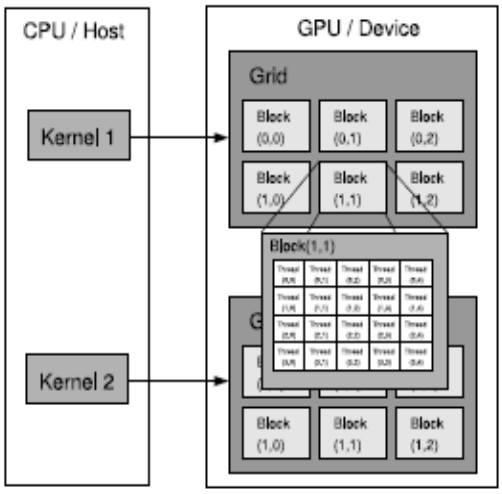

Figure 2: Programming Model
(Source: Nvidia)

4. RESULTS AND ANALYSIS

As an extension to the C language, CUDA provides a high level interface to the programmer. Hence porting algorithms to the CUDA programming model is straight forward. In this paper we report a set of programs which are used to assess the performance of GPU that include array addition, matrix multiplication and FFT. Assessment is based upon the following factors:

- Varying iterations: The core code is put to iterate n number of times and resulting performance is analyzed
- Varying data size: The size of the data structure used is varied and the resulting performance is analyzed.
- Varying block size: Size of the block (i.e. no of threads per block) is varied and the resulting performance is analyzed.

4.1 Array Addition

Arrays find their use in many applications and are most suitable data structure for the implementation of data parallel algorithms. We implement array addition initially using the C programming language, and later implement it on a GPU using CUDA Programming environment. In the later case the program uses one thread per element addition. Both the arrays are mapped to different threads and blocks accordingly to produce an optimal result.

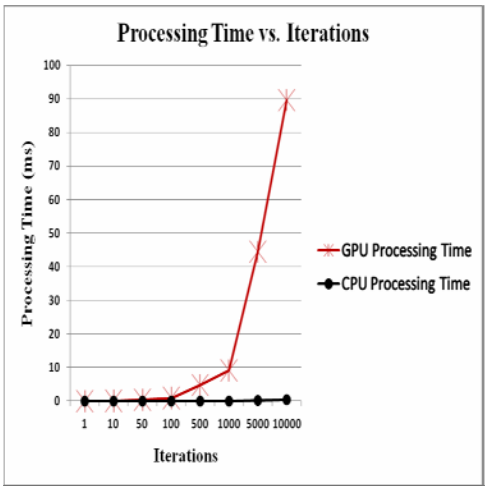

Figure 3: GPU and CPU Times with varying Iterations

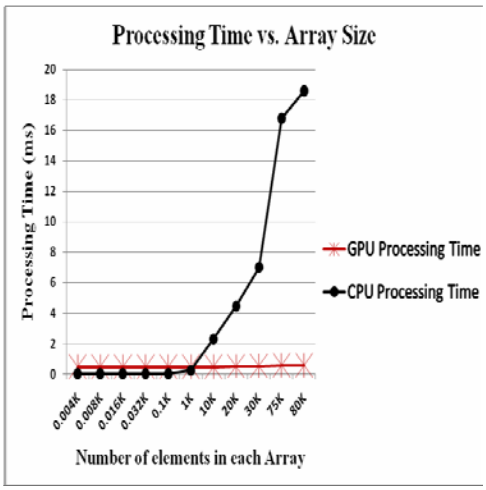

Fig 4: GPU and CPU Times with varying array elements

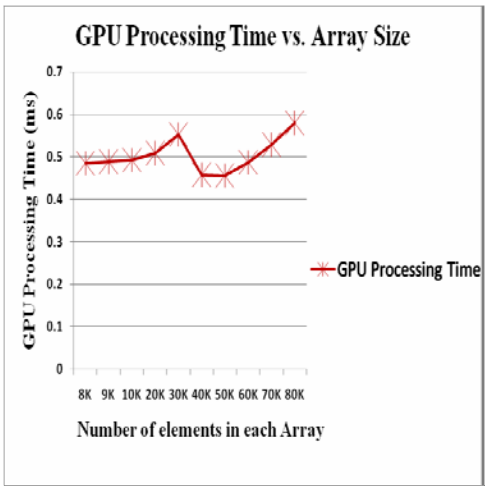

Fig 5: GPU Time with varying array elements

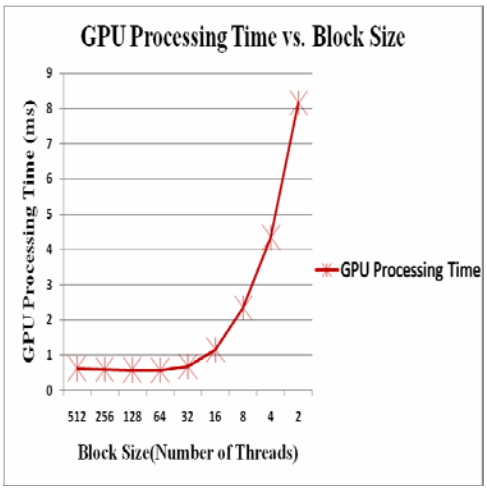

Fig 6: GPU Time with varying Block Size

The experimental results for array addition implementation are shown in fig 3-fig 6, fig 3 shows the varying iterations issue. GPU and CPU time increase exponentially with no of iterations, comparatively CPU shows a less increase in time, in this particular case GPU doesn't show any improvement in the performance as it is restricted by GPU's initial setup time overhead. In fig 4, GPU proves to be a far better choice over CPU in case of varying array size (array elements), for arrays of smaller sizes CPU still shows less time but as the size of the array increases GPU shows a better performance over CPU, for an array of 80K elements the speedup achieved is 31X. Fig 5 shows the GPU time with varying array size, at one particular array size there is a steep fall in GPU time where it gives maximum throughput. Fig 6 shows how GPU time varies with different block sizes (no. of threads). Till 32 threads in a single block GPU time is linear, beyond 32 it starts increasing exponentially.

4.2 Matrix Multiplication

Matrix (2D array) also is an important data structure for the implementation of data parallel algorithms. We have implemented the matrix multiplication program in the following way:
The task of computing the product C of two matrices A and B of dimensions (wA, hA) and (wB, wA) respectively, is split among several threads in the following way:

- Each thread block is responsible for computing one square sub-matrix C_{sub} of C.
- Each thread within the block is responsible for computing one element of C_{sub}.

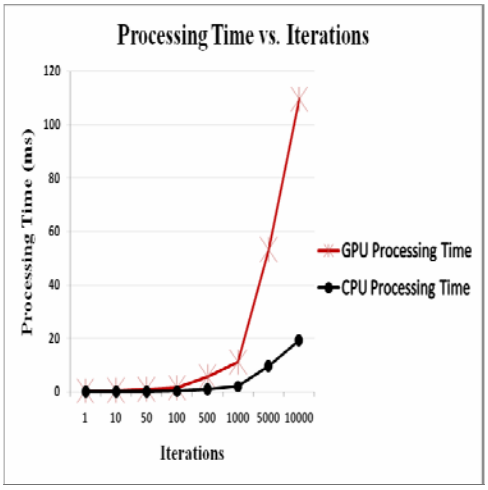

Fig 7: GPU and CPU Times with varying Iterations

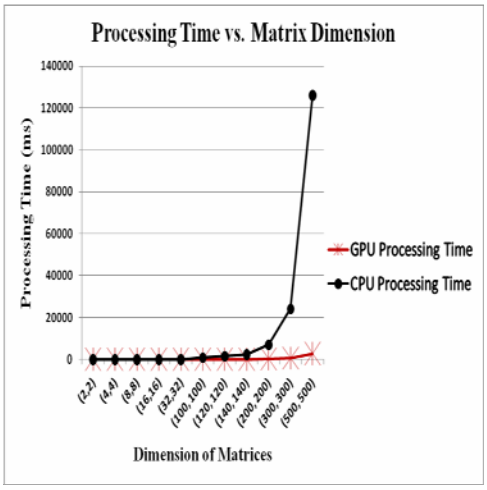

Fig 8: GPU & CPU Times with varying Matrix Dimensions

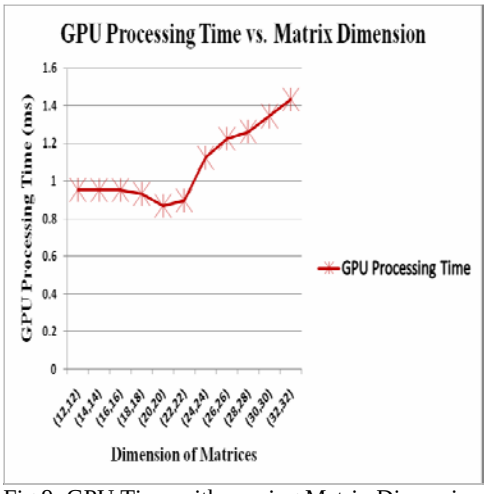

Fig 9: GPU Time with varying Matrix Dimensions

Fig 7-Fig 9 show the experimental results for matrix multiplication implementation. Fig 7 shows varyig iterations issue similar to array addition implementation. Fig 8 again shows GPU's performance enhancement over CPU for matrices of large size. Fig 9 shows the GPU's behavior for different matrix dimensions, for a particular size of matrix GPU gives maximum throughput.

4.3 FFT (Fast Fourier Transform)

The FFT is a divide-and-conquer algorithm for efficiently computing discrete Fourier transforms of complex or real-valued data sets, and it is one of the most important and widely used numerical algorithms, with applications that include computational physics and general signal processing. The CUFFT library provides a simple interface for computing parallel FFTs on an NVIDIA GPU, which allows users to leverage the floating-point power and parallelism of the GPU without having to develop a custom, GPU-based FFT implementation. FFT libraries typically vary in terms of supported transform sizes and data types.

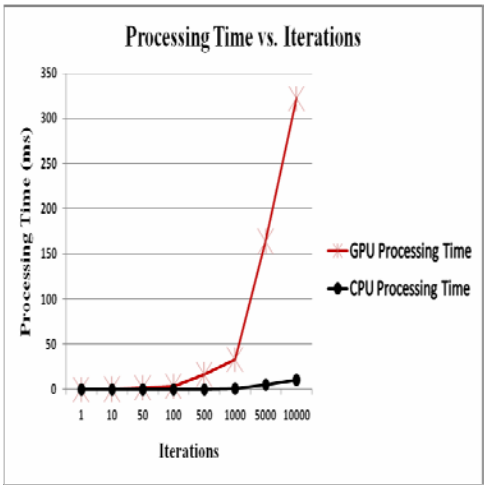

Fig 10: GPU and CPU Times with varying Iterations

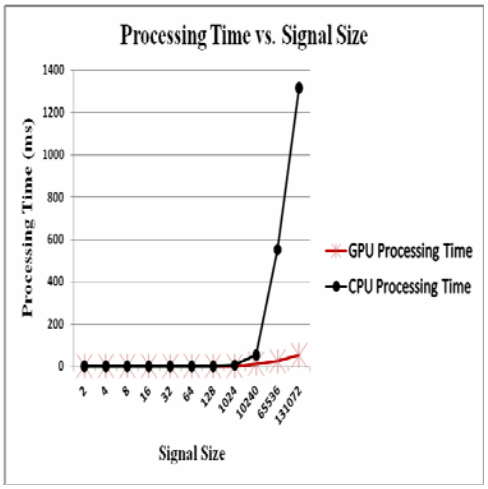

Figure 11: GPU and CPU Times with varying Signal Size

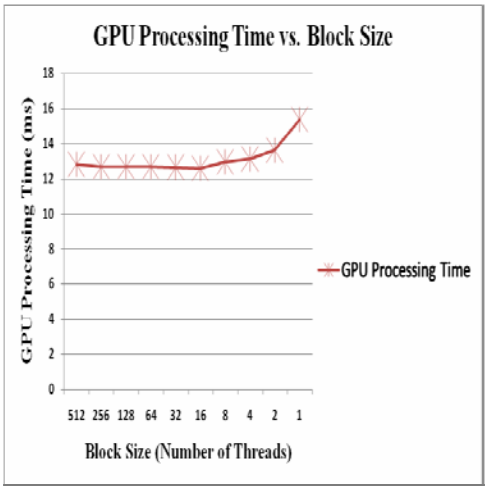

Fig 12: GPU Time with varying Block Size

Fig 10-Fig 12 show experimental results for FFT implementation of both GPU and CPU. For GPU implementation we make use of a built in library called CUFFT and for CPU implementation we use an open source FFT library called fftss. Fig 10 shows the varyig iterations issue which is similar to previous two implementations. Fig 11 shows how GPU improves the performance of FFT when the signal size is very large, GPU time for small signals is more when compared to CPU time, but when the signal size is

large GPU shows considerable improvement in the performance. Fig 11 shows GPU time with varying block size, for almost all block sizes GPU time remains more or less constant, it shows little increase when no of threads become too less for the GPU computation.

5. CONCLUSION AND FUTURE WORK

GPU architectures have evolved to be well suited for solving data parallel problems, and they continue to deliver increasingly higher performance. As a simple extension to the C programming language CUDA provides easy access to high performance GPU computing. In this paper we have presented the analysis of few fundamental SIMD programs on GPU hardware, these programs have wide practical applications. We presented array addition, matrix multiplication and FFT.

The contributions of this research are as follows:

- GPU shows tremendous performance improvement over CPU when the data to be computed is large. On the other hand it doesn't show much improvement over iterations, because in this case GPU's performance is limited by it's initial setup overhead.
- If the size of the data to be computed is less, GPU time gets limited by the data transfer and initial setup time overhead. In such cases CPU proves to be a better choice.
- In order to gain maximum performance from GPU, it is necessary to keep all the multiprocessors of GPU busy with work.
- Launching adequate no of threads in a single block helps in gaining better performance from GPU.
- With suitable changes in the Flosolver's Floswitch and corresponding modifications in the protocols on the Flosolver Parallel supercomputer MK7 which is expected to yield 10TFLOPS with 1024 processors, we assume, with each board having Nvidia GPU, we may expect a tremendous improvement in the speed and performance of MK7 for parallel graphics applications.

In future there is a need for further research in partitioning the problem on multiple GPUs and reducing the memory transfer overhead from CPU to GPU. Another drawback of GPU is the lack of higher precision, a serious limitation for scientific applications. However the use of GPUs as economical, high-performance co-processors can be a significant driving force in the future.

6. ACKNOWLEDGMENTS

The authors wish to acknowledge Dr. U. N. Sinha, Head, Flosolver unit for his invaluable guidance and for his encouragement in completing this paper at every possible step.

7. REFERENCES

[1] Literature Review: High-Performance Computing By Advanced Stream Processing Using Graphics Hardware, Young Sung LEE, seanl@cse.unsw.edu.au, University of New South Wales, Supervisor: Dr. Gabriele Keller, March 14, 2006

[2] NVIDIA_CUDA_Programming_Guide_1.1

[3] Accelerating large graph algorithms on the GPU using CUDA, Pawan Harish and P. J. Narayanan, Center for Visual Information Technology, International Institute of Information Technology Hyderabad, INDIA

[4] Sample Programs running on Graphics card By Shrangar.V, Chetaran.N, Meetei.D.L and Nagendran.S NAL PD FS0725